Mystic: Privacy-focused Data Aggregation in Social Network Platforms

Aparajitha Adiraju
Department of Electrical and Computer Engineering
University of Illinois Urbana-Champaign
Champaign, Illinois
Email: adiraju2@illinois.edu

Arjun Arun
Department of Computer Science
University of Illinois Urbana-Champaign
Champaign, Illinois
Email: arjunsa2@illinois.edu

Priyal Gosar
Department of Electrical and Computer Engineering
University of Illinois Urbana-Champaign
Champaign, Illinois
Email: pgosar2@illinois.edu

***Abstract*—In recent years, traditional social networks have faced criticisms for excessive data collection and lack of user control over data, which in turn has lead to a growth in research on decentralized social networks. On a theoretically completely decentralized social network, data is not accessible from a centralized server, but this can lead to a lot of performance issues / trade offs in practice. In this paper, we introduce Mystic, a decentralized social network where each user only stores copies of their own data on their own device permanently to preserve individual privacy. To enable better performance however, we enable a centralized caching server where posts are stored temporarily based on a user specified TTL (Time-to-live) parameter per post, which allows for users to better tune their privacy controls while still maintaining solid performance for a good end user experience.**

I. Introduction

Social media platforms have gone through many shifts in public opinion over the years. Platforms have gone from media darlings to villains in public perception, largely due to various consequences for their decisions with regards to privacy and the impact of their platforms on the world. With issues ranging from privacy concerns to security issues to even algorithmic bias, there have been a lot of critiques to Facebook, Twitter, and other social media platforms technology. As a response to this, there have been a lot of attempts at decentralized social media platforms. Most of these papers do not focus on the ideas of data aggregation and specifically querying data ephemerally, so this paper couples the ideas of ephemeral data together with the optimizations found in previous work related to aggregation and protocols.

Mystic focuses on a few core ideas to motivate its system architecture. To enable users to own their data, Mystic only stores user data permanently on the user's device itself. For better performance however, we built a central caching server that stores data temporarily and routes it between devices. To ensure that user's still own their data we enable a user specified TTL parameter on every post, which specifies a timestamp that states how long the centralized server should hold the post for once it has been requested. This creates an ideal balance between user control over data and real world performance, and as our results show, in most scenarios Mystic provides similar performance to architectures of traditional networks while also only adding minimal latency and bandwidth increases to the central server in less ideal scenarios.

We summarize the contributions of our paper as the following:

1) Explore architectures and performance of previous decentralized social networks
2) Experiment with different types of caching mechanisms with the trade off between privacy and usability
3) Introduced the novel system architecture Mystic that uses a centralized cache in conjunction with a Time To Live (TTL) expiration on all stored user data (introducing the idea of ephemerally stored data) to enable good performance and better user control on data

than centralized networks

II. Related Work

There are many papers that discuss privacy implications surrounding general social media usage. This paper [8] looks at the privacy implications users on a variety of different social media services face, especially concerning the metadata that is inevitably stored from many user profiles. As a response to general privacy and security concerns, there are a multitude of social media platforms that have been created that will be described in further detail: Diaspora utilizes independently owned servers, Minds focuses on peer to peer advertising and freedom of speech, Mastodon is similar to Twitter, and more described below [4]. A lot of these platforms employ "personalized" servers per user in order to have users be in "control" of their data, but none address the permanence of data.

Encryption is a popular solution for privacy concerns. PrPI is a decentralized infrastructure for online social networking aimed at preserving privacy through the use of individual Personal Cloud Butler services that supports sharing with fine grain control (this can be on a cloud server or even a home server of their choice). Data can be stored on each individual's butler and can reside encrypted on other servers [9]. Another system Reclaim aimed to preserve privacy by utilizing encrypted handshakes to establish "friendships" and secure data transfer with decentralized storage. [14]. Contrail suggests that while decentralized networks can provide privacy, they are difficult to use on devices due to constraints on connectivity, energy, and bandwidth. Through Contrail, users install content filters on their friend's devices that express their interests, and subsequently they receive new data generated by their friends that match these filters. All the data and filters are exchanged via cloud relays that are encrypted, and this thus preserves privacy and is energy and bandwidth efficient [11].

While the above systems focused more on data flow, there are also decentralized social medias that proposed unique server architectures for "safe" relationships. Diaspora launched in 2011, and it consists of a network of independent, federated servers. Diaspora networks allow users to decide which servers users can store information on rather than automating that process on one server or a collection of servers. Users are able to spin up their own private servers to maintain full control, and others can choose to join a third-party owned server [7] Diaspora utilizes a push design to disseminate data across servers to all followers and relies on human trust in administrators of each server to main the the privacy and security of the data, which is not ideal in certain cases.

Mastodon is an open source, decentralized blogging platform similar to Twitter and architecturally similar to Diaspora. The underlying server architecture is implemented with federated data in a decentralized topology, which has a few key servers rather than one central server [6]. User information is stored on instance-based servers, but you can follow users that are a part of different instance servers. All the servers are interconnected as well. Data is aggregated on local and federated timelines: local timelines aggregate data from users from the same instance, and the federated timelines aggregate data from all instances. Users are also able to share information about themselves at different privacy levels. This paper looks into the effects of decentralized social media on the user interactions and flow [5]. Mastodon uses OStatus, which allows federated servers to communicate and is also compatible with GNU Social, which is another protocol standard for federated server communication [16].

CosMeDis is a distributed social media platform that consists of an arbitrary number of communicating nodes, deployable at different locations over the internet. Users can post content and establish intra / inter node friendships, which allow users to regulate control over posts. The paper focuses on security and formalizes a framework for composing a class of information flow security guarantees in a distributed system that is used for posts, friendship requests, and statuses [2]. The CosMeDis privacy and security guarantee is established on strong privacy settings that enable only post admins and creators to decide who views private posts.

While a lot of these social medias address components of the privacy and security issues with multi-hop trust chains, federated data across decentralized servers, personal or "trusted" servers hosting data, all of them rely on the fact that the people and servers that users share their data with are trust-

worthy. In other words, the data is permanent once it is shared and in the hands of followers, server administrators, and others, even if it is encrypted. This is the key distinction that this paper hopes to address. The proposed system architecture Mystic could be used in conjunction with existing social networks to allow for total user control. Joining servers with hundreds of people even with certain privacy settings and personal servers hosting data will allow user data to exist across multiple servers in the system once shared. The concept of data ”disappearing” once shared on the internet does not exist, and this is scary to a lot of internet application users today; not being in control of your own data.

Out of all existing social media platforms, Snapchat has the most similar structure. Their entire platform is based on the ephemeral nature of the messages sent, which can be photos, videos, messages, etc. Snapchat servers delete all unsaved messages after 30 days [19]. The difference between Snapchat and what this paper is proposing is that users' data is not lost if it is not immediately saved.

III. System Model & Architecture

Mystic is a directed social media platform with users that can ”follow” other users and view their follower's posts. Each user has a set of ”followers”, users that follow them, and ”following”, users that they follow. A user can make a post that consists of a maximum of 200 characters and 4 attachments (files such as pictures, etc). When it is posted to their *timeline*, a log of all the posts of a specific user, it will be available for all of their followers to see. A user can view any of the posts of the users they follow by sending a timeline request to the central server. When a post is made, it is sent to a central server with a user specified TTL, and after that time has passed the post data is deleted from the central server and will only live on the user's device. If a follower requests to see a user's post and it has expired on the central server, then the server will make request(s) to the specific user for the data and it will again be stored on the central server with a new TTL timestamp.

Some of the major design requirements of Mystic included a decentralized permanent storage of data and a way to remove data from the central server, while still providing a standard social media experience on the front-end. In order to achieve these requirements, Mystic makes the following contributions:

1) A unique data flow for user's posts where the data from a user's posts are sent to a central server and stored there temporarily.
2) Data for a certain user's post only exists permanently on the user's local device. If the data on the central server disappears, or becomes stale, then the central server must re-request the data from the user's local storage.
3) Mystic also has the basic functionality for users within the system to post, follow other users, and view other's posts, similar to Twitter's set-up.

To understand the novelty of data flow in Mystic, we observe a typical social media platform's data flow.

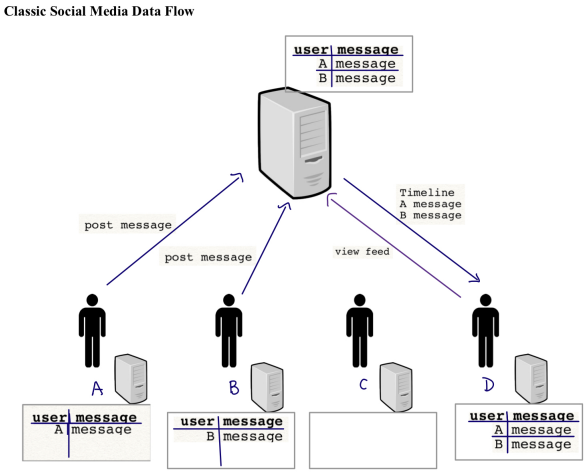

Fig. 1. Data flow in typical social media

In *Fig. 1*, User A and User B both post messages. User D follows both User A and User B and refreshes their feed views both User A and User B's messages. There is a central server, or a cluster, that stores messages (posts) that all the users post. This data is often never deleted off the main server and is instead stored permanently across the central servers once any message is posted. When another user requests another user's message data (by either ”refreshing” their feed or going to the user's personal timeline), the data is fetched from the main server and then sent directly back to the user requesting the information.

In Mystic, we have the following data flow:

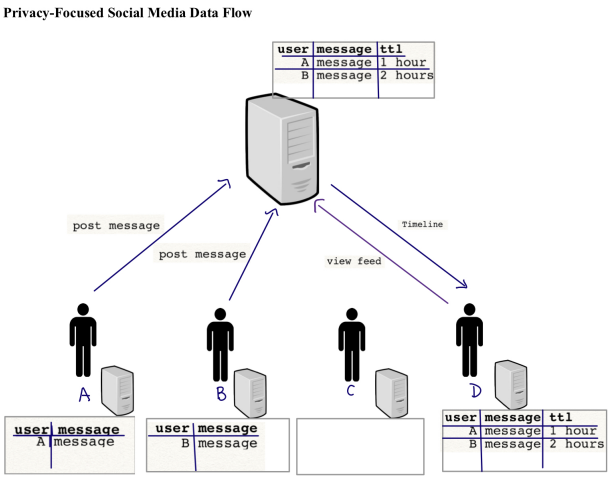

Fig. 2. Data Flow in Mystic

In *Fig. 2*, we have the same scenario where User D is viewing User A and User B's messages on Mystic. The major difference between Mystic and typical social media is that each message by a user that exists anywhere except on the user's device has a TTL. This means that after the TTL elapses, the message data is deleted from the central server and any other devices the data was saved on. If a user wants to view the data and it has already expired on the central server, then the central server will fetch the data from the user that originally posted it and forward the data to the requesting user.

The following example further describes Mystic's behavior.

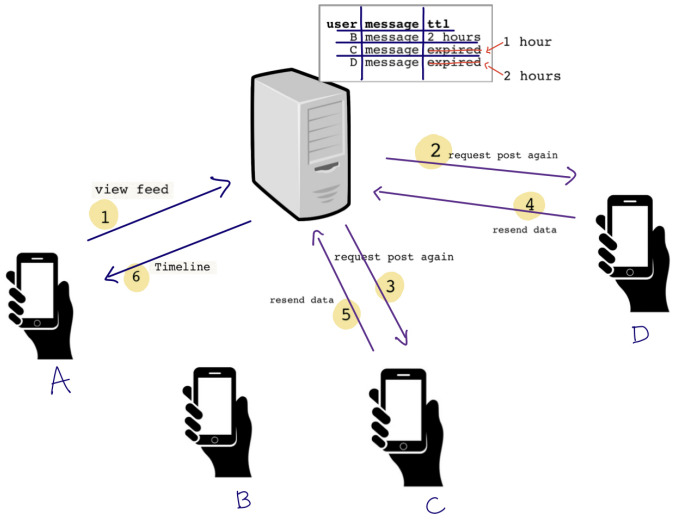

Fig. 3. Re-requesting data in Mystic

In Step 1, User A follows User B, C, and D and sends a request to the central server to view their feed. At the central server however, the messages from C and D have expired and therefore the central server will re-request the messages from Users D and C (Steps 2 and 3). Once they resend the data to the central server (Steps 4 and 5), their TTLs are reset and the central server forwards the information to User A. In this way, the main trade off between traditional social media and Mystic's data flow is primarily the latency to retrieve expired data. However, this TTL feature also adds more security and privacy as the data is only permanently stored on the user's local device.

It's important to note that there is metadata related to posts that is stored on the central server permanently, such as the post id and the user id of the post, but the contents of the post and attachments related to the post are deleted according to the respective TTL on the post. This allows the central server to be able to route post data effectively from different user devices.

IV. Implementation

Mystic is implemented in the Go programming language. Mystic has 2 distinct logical components: server / client logic for the central server, and separate server / client logic for user devices. To communicate between the central server and the user devices we created a universal protobuf struct Message, which is outlined in detail in Figure 4.

Struct *Message*:

```
// Central Server: 0 is join, 1
   for follow, 2 for unfollow, 3
   for post, 4 for view, 5 for
   resent post
// Client: 0 is timeline posts,
   1 is a request for a post
   based on id
uint32 type = 1;
 string username = 2;
 repeated Post posts = 3;
 string optional = 4;
```

end

Fig. 4. Message Struct in Mystic

The central server continuously runs and accepts messages from various user devices. The central server has operations allowing it to deal with users joining the network, follow and unfollow others, post content, view timelines, and handle resent posts that have expired. This is specified based on the type field shown in the Message struct in Figure 4. The central server uses a dictionary to map user ids to a set of user posts, which allows it to quickly look up and find the most recent posts necessary to fill the timeline request of any specific user based on their ”following” list. We use the posts field in the Message struct to send back the posts of a specific users timeline back to the user device. In Figure 5 we can see the Post struct protobuf which outlines how posts are handled in our system, where we keep track of a post id, the TTL associated with it, the textual data of the post, and then also have a repeated string field that can represent file paths that represent attachments to a post.

The user device is currently implemented through a command-line input (CLI), in which it continuously scans for user inputs for the commands of join, follow, unfollow, post, view, and more. With any of these requests, the user device will send the appropriate message with the necessary details to the central server, which will route or store data as necessary to ensure the proper end user

Struct *Post*:

```
string id = 1;
uint64 ttl = 2;
string data = 3;
repeated string attachments = 4;
```

end

Fig. 5. Post Struct in Mystic

experience. To enable the central server to ask for expired posts on demand, we enable a continuously running server on the user device as well, which can accept requests and send the appropriate posts back to the central server asynchronously. All client functionality is implemented as function endpoints, and the server functionality handles each of those requests. Example function calls and endpoints from both the client and server perspective can be seen below in figures 6 and 7 as well.

Go was chosen as the programming language for Mystic due to the ease of use of GoRoutines which allowed for ease of use of threads and parallelism which was able to optimize performance throughout. Beyond that, in conjunction with protobufs, the networking and message passing of our system was able to be implemented efficiently and reliably.

Our implementation source code can be found in the GitHub Repo here: https://github.com/arjun75x/mystic

A. Implementation Assumptions

1) Users each have a server running continuously to be able to access data whenever possible. If a user's server is down and the posts that another user wants have expired on the central server, then that user will simply not be able to view the post.
2) If user's have multiple devices then only 1, ideally the continuously running server mentioned above, will store the data permanently and the rest will act like clients.

Algorithm 1 Send A Follow Message

```
msg = CreateMessage("Follow", username)
SendMessage(msg)
```

Fig. 6. Pseudocode for following another user in Mystic client side

Algorithm 2 Handle Follow Message

```
add new follower to userFollowers[user] set
```

Fig. 7. Pseudocode for following another user in Mystic server side

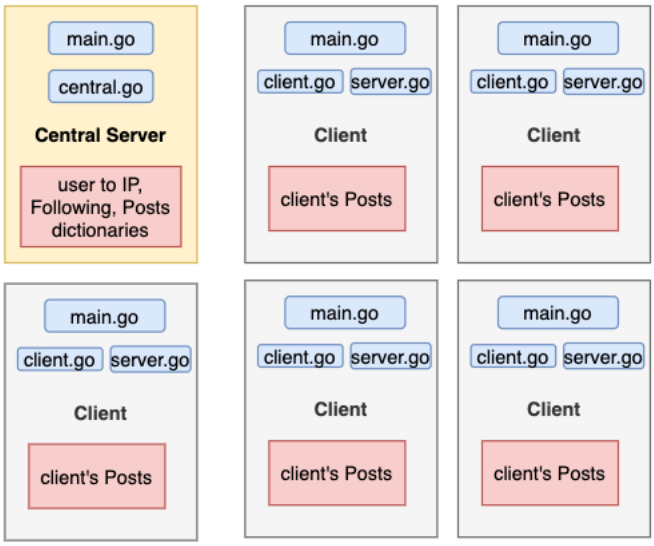

Fig. 8. Mystic Infrastructure running on six VMs

V. Experiments & Results

Ideally, Mystic would be tested alongside existing social networks in order to judge the key feature of privacy. However, running tests alongside Facebook or even Mastodon would not be scientific, given that latency, bandwidth, and other calculations would essentially be meaningless with vastly different numbers of users and underlying server infrastructures.

For our experiments, we ran Mystic across a testbed of 10 VMs that are all running the same executable. We hardcoded in 1 VM as the central server and assumed all the 9 other VMs were aware of its IP address beforehand. Device characteristics were not taken into account when creating this system. All experiments were conducted on VMs as described, but ideally this system would be expanded to different types of devices (laptops, mobile devices, etc).

The Client VMs have the ability to "join" the system, "follow" and "unfollow" other users on the system, "post" data, and "view" their timeline (which aggregates data from all of their followers). For testing and comparison purposes, the TTL and ephemeral nature of "posted" data was parameterized so results could easily be compared between the two systems.

To test the performance of our system relative to a normal centralized social network, we focused our experiments on the scenario where a user makes a timeline request to the central server and the data is available versus data is expired. This allowed us to test the performance of our re-requesting and routing logic, which is the core difference between Mystic and traditional social media. In situations where the user requested timeline data is completely available on the central server, Mystic will perform just as well as centralized social media.

A. Experimental Assumptions

1) Experiments are limited across 10 machines with a hardcoded port number for the sake of initial experimental data. Users each have one IP and one port.
2) The "posts" are implemented through text structs which also store optional filepaths to indicate attachments. Information is saved on client-side in a specified folder which serves as an the attachment database.

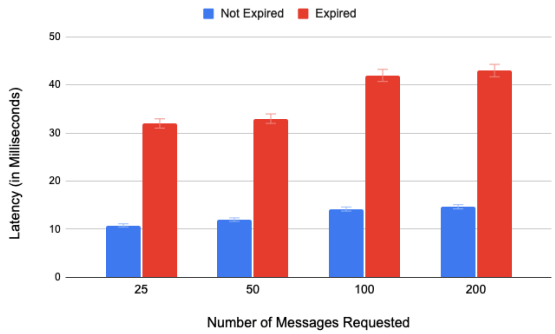

Fig. 9. Latency of timeline request VS Number of messages requested

In Figure 9, we can see that when data is expired on the central server we see an increase in latency when compared to the data not being expired, which is solely due to the re-requesting and re-routing of data happening behind the scenes. These latency numbers were measured in milliseconds, and in actual practice the notice was not noticeable since the differences were only on the orders of 10s of milliseconds.

An important trend to observe is that as the number of expired messages increases, the latency doesn't noticeably increase, meaning that the latency doesn't scale linearly with the number of expiring messages. This is due to the fact we implemented all our request logic in parallel through goroutines, so the central server is able to aggregate missing data as quickly as possible in parallel, with only a slight overhead. This means that the largest size message that needs to be re-requested acts as a bottleneck on the latency of our routing logic.

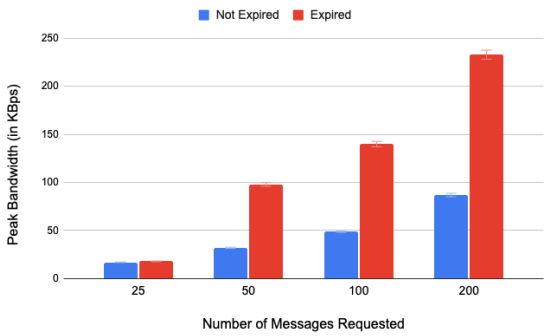

Fig. 10. Peak outgoing bandwidth of central server vs Number of messages requested

In Figure 10, we can see the outgoing peak bandwidth of the central server increases as well when data is expired, since the central server has to send N requests out for new data, then once it has it stored on the servers it has to send N pieces of data out. Since we are sending all our data in parallel at once, this creates a non-negligible increase in our peak bandwidth when sending out all the post data back to the user devices. When scaling the system in scenarios with large amounts of user devices making timeline requests at once, this peak bandwidth number should be studied more closely since we don't want to overload the central server's core network.

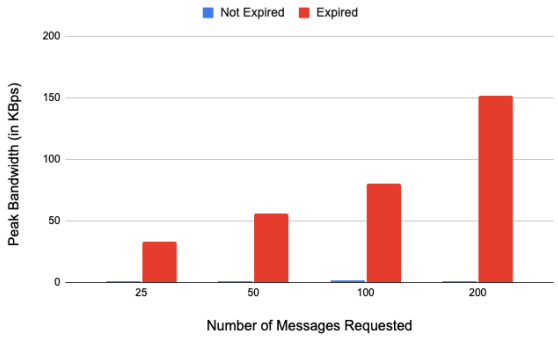

Fig. 11. Peak incoming bandwidth of central server vs Number of messages requested

Lastly, in Figure 11, we can see that for the scenario where data isn't expired the peak incoming bandwidth is negligible at around 1 kbps, since there is only 1 request coming in for data and the central server doesn't have to take in anymore data. However, we see again that when data is expired our peak bandwidth scales linearly with the number of messages re-requested.

In summary, our experiments show that for the end user, the experience of Mystic performs just as well as centralized social media in most scenarios (when data is still cached on the central server), and even when data needs to be re-requested the latency of the timeline view request is not increased significantly and also doesn't scale linearly with the number of posts missing, which is especially good for inactive users who may have lots of posts that are expired. Lastly, we also see that the main impact of our re-requesting and re-routing logic is seen in the peak bandwidth of central server operations which grows linearly with the number of posts missing.

VI. Future Work

A. Edge Servers and Cache Hierarchies

This architecture can be extended by adding Edge Servers which can act as cache hierarchies together with the central server. This can allow for better performance for the end user since data can be stored even closer to them and accessed more quickly. There are a lot of additional performance tweaks that can be added to this solution as well, such as different TTLs for different edge layers, and the routing between Edge and Central servers can be optimized further to improve the performance of our system by reducing the overall latency the end

user sees. This would involve more complex routing algorithms.

This could also be implemented using structures that prefetch data knowing that the TTL will expire rather than on on the timeline view request, which may speed up the user's viewing timeline latency as well. More complex routing algorithms could utilize each individual link bandwidth to determine which link would be the fastest to request, so even if the data being requested is not that user's data, they have it and user requesting the data is able to see it. However, this would drastically change the architecture of the current system given that an additional layer of "prefetching" servers would need to be implemented in order to successfully reduce the latency in fetching data, or users would have to consent to having their data be visible to followers even if not directly requested (like "requesting timeline"). This cache-memory concept could be useful given certain request patterns and warrants future work. This could be implemented with a TTL monitor to warn the system of very low TTLs.

B. Fault Tolerance

In this implemented system, there is a centralized server that routes all the requests from all clients. In the case of a server failure, the current system would not be able to recover any requests that were being serviced and would lose all metadata stored on the client. In theory, this could be handled through some sort of fault-tolerance algorithms that exist [17, 18].

Beyond that, user data is stored permanently only on the user's device itself, and currently it is solely the responsibility of the user's to ensure their data isn't lost, which means that the user's devices themselves need to be fault tolerant.

C. Underlying Decentralized Architecture

One paper related to sensor networks (distributed event based systems with many more constraints than on normal servers) studied the various trade offs related to data aggregation in a distributed setting, and find that the complexity of optimal data aggregation is an NP-hard problem in general, but there are a few useful polynomial time special cases that can be exploited [3].

D. Encryption & Ephemeral data

As described above, existing social media networks utilize encryption as a method of protecting data. It may be interesting to examine and evaluate data conditionally to determine what data should not be ephemeral (profile information or extremely large messages with very short TTLs) and encrypt any data that permanently resides on other user's devices and servers. However, with the addition of encryption to the privacy suite, key management becomes another issue to solve, and truly private schemes would have users manage their own tokens.

E. Other Areas

A few other networks we looked at focused on different problems. For example, Armor explores friend suggestions using social relationships and attributes and uses a privacy-preserving protocol to suggest friendships using multi-hop trust chains between users [15]. Another paper addresses the issue of privacy and data flow by using social convention and context help determine how incoming and outgoing information flows between "producers" and "consumers". The paper suggests also incorporating privacy in these decentralized social networks by assigning "actors" in the system appropriate rights [12]. The problem of where data should be stored is addressed by IPFS as a distributed data storage system and blockchain technology in another paper. This has a notion of a "user contract" which sends IPFS requests and retrieves data from there to fill the content of their proposed decentralized social network [13].

VII. Conclusion

Privacy in data storage and propagation is an important issue considering the ubiquitous nature of social media in today's society. With the rise of social media applications and the rampant data collection and storage that comes along with it, it is important to know where data is being stored and how it may be shared. This paper addresses the issue of storing data permanently across all types of social medias (both centralized and decentralized) and introduces Mystic, a system that allows for ephemeral data storage with some latency overhead added. The use of a TTL parameter allows for better user control of data while also maintaining

solid overall performance. Lastly, our experiments show that the re-requesting and re-routing logic created for expired data adds a fixed latency to requests bounded by the size of the largest missing post, and increases the peak bandwidth load on the central server linearly based on the number of posts missing.

ACKNOWLEDGMENT

The authors would like to thank Professor Indranil Gupta and the CS 525 Staff for making CS 525 a conducive environment to conduct research and explore distributed systems.

REFERENCES

[1] Dorsey, J. (n.d.). https://twitter.com/jack/status/12047660784689111

[2] T. Bauereiß, A. P. Gritti, A. Popescu and F. Raimondi, "CoSMeDis: A Distributed Social Media Platform with Formally Verified Confidentiality Guarantees," 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, 2017, pp. 729-748, doi: 10.1109/SP.2017.24.

[3] L. Krishnamachari, D. Estrin and S. Wicker, "The impact of data aggregation in wireless sensor networks," Proceedings 22nd International Conference on Distributed Computing Systems Workshops, Vienna, Austria, 2002, pp. 575-578, doi: 10.1109/ICDCSW.2002.1030829.

[4] Top 5 Decentralized Social Networks. (n.d.). TechRepublic. https://www.techrepublic.com/article/top-5-decentralized-social-networks/

[5] Zignani, M., Gaito, S., Rossi, G. P. (2018). Follow the "Mastodon": Structure and Evolution of a Decentralized Online Social Network. Proceedings of the International AAAI Conference on Web and Social Media, 12(1).

[6] Zulli, D., Liu, M., Gehl, R. (2020). Rethinking the "social" in "social media": Insights into topology, abstraction, and scale on the Mastodon social network. New Media Society, 22(7), 1188–1205. https://doi.org/10.1177/1461444820912533

[7] A. Bielenberg, L. Helm, A. Gentilucci, D. Stefanescu and Honggang Zhang, "The growth of Diaspora - A decentralized online social network in the wild," 2012 Proceedings IEEE INFOCOM Workshops, Orlando, FL, USA, 2012, pp. 13-18, doi: 10.1109/INFCOMW.2012.6193476.

[8] M. Smith, C. Szongott, B. Henne and G. von Voigt, "Big data privacy issues in public social media," 2012 6th IEEE International Conference on Digital Ecosystems and Technologies (DEST), Campione d'Italia, Italy, 2012, pp. 1-6, doi: 10.1109/DEST.2012.6227909.

[9] Seong, S., Seo, J., Nasielski, M., Sengupta, D., Hangal, S., Teh, S.K., Chu, R., Dodson, B., Lam, M. (2010). PrPl: a decentralized social networking infrastructure. MCS '10.

[10] G. Mega, A. Montresor and G. P. Picco, "Efficient dissemination in decentralized social networks," 2011 IEEE International Conference on Peer-to-Peer Computing, Kyoto, Japan, 2011, pp. 338-347, doi: 10.1109/P2P.2011.6038753.

[11] Stuedi P. et al. (2011) Contrail: Enabling Decentralized Social Networks on Smartphones. In: Kon F., Kermarrec AM. (eds) Middleware 2011. Middleware 2011. Lecture Notes in Computer Science, vol 7049. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-25821-33

[12] G. Groh and S. Birnkammerer, "Privacy and Information Markets: Controlling Information Flows in Decentralized Social Networking," 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing, Boston, MA, USA, 2011, pp. 856-861, doi: 10.1109/PASSAT/SocialCom.2011.30.

[13] Q. Xu, Z. Song, R. S. Mong Goh and Y. Li, "Building an Ethereum and IPFS-Based Decentralized Social Network System," 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS), Singapore, 2018, pp. 1-6, doi: 10.1109/PADSW.2018.8645058.

[14] Zeilemaker, N., Pouwelse, J. (2014). ReClaim: a Privacy-Preserving Decentralized Social Network. FOCI.

[15] Ma, X., Ma, J., Li, H., Gao, S. (2018). ARMOR: A trust-based privacy-preserving framework for decentralized friend recommendation in online social networks. Future Generation Computer Systems, 79(0167-739X), 82-94. doi:https://doi.org/10.1016/j.future.2017.09.060

[16] Bradbury, D., Ducklin, P. (2017, April 10). Mastodon: New beast to challenge big social, or another white elephant? Retrieved March 01, 2021, from https://nakedsecurity.sophos.com/2017/04/07/mastodon-new-beast-to-challenge-big-social-or-another-white-elephant/

[17] Arif Sari, Murat Akkaya (2015) Fault Tolerance Mechanisms in Distributed Systems. International Journal of Communications, Network and System Sciences,08,471-482. doi: 10.4236/ijcns.2015.812042

[18] Sari, A., Akkaya, M. (2015). Fault tolerance mechanisms in distributed systems. International Journal of Communications, Network and System Sciences, 8(12), 471.

[19] Snapchat Support. (n.d.). https://support.snapchat.com/en-US/article/when-are-snaps-chats-deleted.